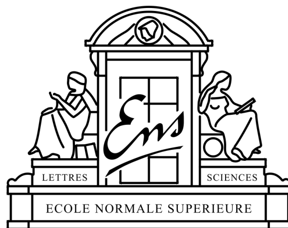
Verification of optimistic algorithms using temporal logic

Guillaume Claret
Département d'Informatique
École Normale Supérieure
45 rue d'Ulm, Paris, France
`guillaume@claret.me`

Zhong Shao
Flint Groupe, Yale University
51 Prospect Street
New Haven, CT, USA
`shao-zhong@cs.yale.edu`



September 2010

Abstract

Optimistic algorithms are lock-free concurrent algorithms, which rely only on a set of atomic operations to provide hight performances. But they can be extremely hard to reason about, and can have really tricky bugs. Many logical systems where built to verify them, mostly based on separation logic, but with heavy use of history variables. Our team just proposed a *Program Logic for History* (by *Ming Fu*), based separation logic, rely guarantee reasoning and temporal operators over traces. It prevents the need of history variables, and allows more intuitive proofs.

I will introduce this logic, and show how to apply it to provide a safety proof of the optimistic queue algorithm of *Michael* and *Scott*. I will also present our *Coq* implementation started with *Yannis Kouskoulas*. Its aim is to both verify the soundness of the rules, and provide a framework to formalize and check optimistic algorithms.

Contents

1 **Introduction** **3**

2 **Work on compilers** **3**
- 2.1 Proof Carrying Code . . . 3
- 2.2 Verified compiler . . . 3

3 ***CompCert* C compiler** **4**
- 3.1 Introduction . . . 4
- 3.2 Compiler chain installation . . . 4
- 3.3 The incrementation instruction . . . 4
- 3.4 External function calls . . . 4

4 **Toy language with in-lined assembly** **5**

5 **Optimistic concurrency** **5**

6 **Queue example** **5**
- 6.1 Choice of the algorithm . . . 5
- 6.2 Algorithm description . . . 6
- 6.3 Safety property . . . 7
- 6.4 Safety proof . . . 8
  - 6.4.1 General properties . . . 8
  - 6.4.2 Proof . . . 9
- 6.5 Real implementations . . . 12
- 6.6 Future work . . . 12

7 ***Coq* implementation** **12**
- 7.1 Code base and first improvements . . . 12
  - 7.1.1 Other separation logic implementations . . . 13
  - 7.1.2 Syntax trees of assertions . . . 13
  - 7.1.3 Logical variables . . . 13
  - 7.1.4 Equivalence of assertions . . . 14
  - 7.1.5 Non-empty traces . . . 14
- 7.2 Future work . . . 15

8 **Conclusion** **15**

9 **Acknowledgments** **15**

1 Introduction

I did my internship in the Flint group at Yale University, directed by professor *Zhong Shao*. The goal of our group is to explore new ways to develop tools to verify complex programs, such as operating systems and concurrent algorithms. There is my homepage where I put what I did each week.

The first three months I worked on many subjects, such as operating systems, file systems and compilers. I read several papers and attempted some classes, but it was hard for me to focus on a specific and concrete subject. I mostly worked on *C* compilers and *Proof Carrying Code*, trying to combine proofs on hight level languages with proofs on assembly code in an efficient and modular way, for operating system verification.

During the second half, I switched to work on our new temporal logic. This logic was presented just this September at *Concurr 2010* in Paris, so we needed to apply it to some problems. Our goal is to use it to simplify and formalize most of the proofs made on optimistic programs. We may have also to extend it to be able to prove the *safety*, but also the *liveness*, the *linearizability*, ... It is also important to check the logical rules with a proof assistant, which can also allows to formalize programs' proofs.

2 Work on compilers

The goal is to be able to mix assembly and C code, in order to prove low-level programs, such as operating systems. There are different kind of ways to explore to achieve that.

2.1 Proof Carrying Code

When doing PCC, proofs are usually made on assembly code. But we can also do them on C, assuming a compiler able to translate C assertions and proofs to assembly ones.

A lot of PCC framework and program semantics have been proposed. A mature framework is OCAP, and I am currently reading the associated paper. The target semantics should be Hoare and Separation Logic.

I did not yet investigated compilers with assertions translations.

2.2 Verified compiler

Some projects, such as *CompCert*, aim to produce a compiler for C, verified in the sense that there is a proof that they keep program semantics during compilation. We would like to do the same thing using both C and assembly.

CompCert is not the only one verified compiler : there is also Concurrent CompCert, or also compilers for ML / Java. We can also try

to reason about toy or abstract compilers. Assembly code can be or in-lined or called as an external function.

3 *CompCert* C compiler

3.1 Introduction

CompCert is a verified *C* compiler, started by *Xavier Leroy*, written both in *Coq* and *Caml*. This means that if a compilation successes, the generated assembly code will have the same *observational* semantics as the source code (the same list of input / ouput). Most of the *C* language is handled, and all standard optimizations are made. Several intermediate languages are used thought all the process, all sharing the same observational behavior.

3.2 Compiler chain installation

Unfortunately, *CompCert* produce only PowerPC assembly.

To use it on an x86 processor, I had to install a cross compiler version of *GCC* using scripts of Dan Kegel. Thus, PowerPC assembly can be compiled to PowerPC binaries. And, thanks to *Qemu* user mode emulation, these binaries can be run on x86 as any normal program. You only need not to use external libraries, and compile with `-static` option.

3.3 The incrementation instruction

As a test, and to try the different layers of *CompCert*, I added an incrementation operation to the compiler. This task was easy, since this instruction only compute on registers, and does not change the memory organization for example.

I added this instruction to the `Op` sum type, which describes available basic assembly operations. It is used by the different compilation layers, except for the back-end and front-end. Most of the time, translation from an intermediate language to another was made even without modifications for `Op` based expressions. But some proofs had to be copied in different places, whereas they were similar to the ones for other operations : more factorization could be possible.

What was not done is instruction selection optimization (not our interest) and the part of the front-end to use incrementation `var++` of C. Moreover, this extension is not very scalable, since the `Op` type is specific to a processor architecture (here, PowerPC).

3.4 External function calls

A C program has to do external function calls, for system calls or modular compilation with libraries. In *CompCert*, the list of external calls even define the common semantics of C and assembly. They are mainly described in `Events.v`.

External functions are assumed to be of a restricted set of kinds, even if it is not the case :

- *syscall* : do not change the memory ; take arguments and return a result
- *load* / *store* : memory operations
- *malloc* / *free* : heap operations

Their common semantics is expressed as a small-step relation :

$$(arguments, mem) \mapsto (result, mem')$$

Hence, all external call verify a set of properties, such as well typing, determinism, and that the memory is still well-formed. If an external library written in assembly verify these properties, then we can compile code calling this library keeping the observational behavior.

4 Toy language with in-lined assembly

I also worked on C programs with in-lined assembly. The key point is to have a convention, so the two languages can communicate values, from registers to variables, and from variables to registers. Next, we can define a semantics of the source code, and show the equivalence.

I worked on some toy languages, and started a formalization here.

5 Optimistic concurrency

Optimistic concurrency is hard to verify, because programs are mostly based on the low-level `CAS` instruction instead of the `lock` :

$$CAS(a, b, c) \;=\; if(a == b)a = c;$$

So besides the use of tools such as *separation logic* and *rely-guarantee*, it is important to keep track of the past events.

This can be achieved using historical variables (pseudo-variables introduced in code annotations to keep track of events), or as in the *Fu*'s paper with temporal operators. These operator are applied on the trace of previous states, recorded with a trace semantics.

6 Queue example

We are trying to apply Ming Fu's Historical Logic to give a formal safety proof of an optimistic queue algorithm.

6.1 Choice of the algorithm

Multiple optimistic queue algorithms were proposed in the past. The Michael and Scott algorithm stands as a reference, but the paper only gives an informal proof.

They are two versions : one with a GC, simpler, and one without, more subtle and which requires to add a modification counter to each pointer. The GC version already has a formal proof in a paper from Lindsay Groves, so we will focus on the more interesting version without GC.

6.2 Algorithm description

The queue is made of a simply linked list of nodes, with two pointers : *Head* to a dummy head node and *Tail* to the last or second last node. All the pointers have a modification counter field.

```
pointer = {
node * ptr;
int c;
}
node = {
int value;
pointer next;
}
queue = {
pointer Head;
pointer Tail;
}
```

Data structures

```
initialize() {
node * node;
node->next.ptr = NULL;
Head = Tail = node;
}
```

Empty queue

```
void enqueue(int value) {
01:  node * node;
02:  node->value = value;
03:  node->next.ptr = NULL;
04:  while(true) {
05:    tail = Tail;
06:    next = tail.ptr->next;
07:    if(tail == Tail) {
08:      if(next.ptr == NULL) {
09:        if(CAS(&tail.ptr->next, next, (pointer)(node, next.c+1))) {
10:          CAS(&Tail, tail, (pointer)(node, tail.c+1));
11:          return;
12:        }
13:      } else
14:        CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1));
15:    }
```

```
16:  }
}
```

Enqueue

```
bool dequeue(int * pvalue) {
01:  while(true) {
02:    head = Head;
03:    tail = Tail;
04:    next = head.ptr->next;
05:    if(head == Head) {
06:      if(head.ptr == tail.ptr) {
07:        if(next.ptr == NULL)
08:          return FALSE;
09:        CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1))
10:      } else {
11:        *pvalue = next.ptr->value;
12:        if(CAS(&Head, head, (pointer)(next.ptr, head.c+1))) {
13:          free(head.ptr);
13:          return TRUE;
14:        };
14:      }
15:    }
16:  }
}
```

Dequeue

Note that there are a lot of dangerous dereferencings, like l.6 in *enqueue*, which could fail if the memory region has been freed. So we need some specific hypothesis on the *malloc* / *free* functions (see the ROP problem).

6.3 Safety property

We will assume there are several threads indexed by tid_1, tid_2, ... Each will execute an *enqueue* or a *dequeue* on one common queue structure, described by $Head$ and $Tail$.

Here are the predicate used to define a queue :

$$
\begin{aligned}
goodtriple(L_l, L_v, L_p) &= length(L_l) = length(L_v) = length(L_p) \neq 0 \\
list([l],[v],[p]) &= l \mapsto (v,p) \wedge (p.ptr = null) \\
list(l : L_l, v : L_v, p : L_p) &= goodtriple(L_l, L_v, L_p) \wedge (p.ptr = L_l[0]) \wedge \\
&\quad\; l \mapsto (v,p) * list(L_l, L_v, L_p) \\
headtail(L_l) &= lenght(L_l) \neq 0 \wedge (H = L_l[0]) \wedge \\
&\quad\; (T = L_l[-1] \vee (length(L_l) \geq 2 \wedge T = L_l[-2])) \\
queue(L_l, L_v, L_p) &= headtail(L_l) \wedge list(L_l, L_v, L_p)
\end{aligned}
$$

We always suppose that L_l, L_v and L_p are not empty and have the same length. L_l are locations, L_v `int` values and L_p pointers with

counter. $L[0]$ means the first element of a list, $L[-1]$ the last one, $L[-2]$ the second last, and $L[m..n]$ a sub-list from element m to n.

We define the atomic operation each thread can do on the queue :

$$\begin{array}{rcl} enqueue_{tid} & = & \exists L_l, L_v, L_p, v, l, \\ & & \quad queue(L_l, L_v, L_p) \ltimes_{tid} \\ & & \quad queue(L_l + [l], L_v + [v], L_p[0..-2] + [(l, L_p[-1].c + 1), (null, _)]) \\ dequeue_{tid} & = & \exists l, v, p, L_l, L_v, L_p, \\ & & \quad queue(l : L_l, v : L_v, p : L_p) \ltimes_{tid} \\ & & \quad queue(L_l, L_v, L_p) \\ changetail_{tid} & = & \exists L_l, L_v, L_p,\ queue(L_l, L_v, L_p) \ltimes_{tid} queue(L_l, L_v, L_p) \end{array}$$

The last operation is not exactly Id, because the $Tail$ pointer can change.

Some definitions about pointers :

$$\begin{array}{rcl} keep(x) & = & \exists X, (x = X) \ltimes (x = X) \\ pointer(p) & = & \forall C,\ (\ominus p.c = C) \Rightarrow (keep(p) \vee p.c = C + 1) \\ pointers & = & \forall L_p,\ queue(_, _, L_p) \Rightarrow \\ & & (pointer(Head) \wedge pointer(Tail) \wedge (\forall p \in L_p, pointer(p))) \end{array}$$

which means that all the pointer counters are strictly increasing during updates.

The R, G, I properties :

$$\begin{array}{rcl} I & = & \exists L_l, L_v, L_p,\ queue(L_l, L_v, L_p) \\ G_{tid} & = & (enqueue_{tid} \vee dequeue_{tid} \vee changetail_{tid}) \wedge (I \ltimes_{tid} I) \wedge pointers \\ R_{tid} & = & \bigvee_{tid' \neq tid} G_{tid'} \end{array}$$

We omit invariants and properties about local variables of each process.

6.4 Safety proof

6.4.1 General properties

To simplify the proof, we will use some properties on the data structures which come from the invariants.

- If a pointer p has the same value twice, counter included, it means that it was a constant (here, P means a constant pointer value, φ a formula) :

$$\left.\begin{array}{c} \boxminus\, pointer(p) \\ (p = P) \blacktriangleright \varphi(P) \\ p = P \end{array}\right\} \Rightarrow \diamondminus \varphi(p) \qquad (1)$$

- If $Tail$ points to the end of the queue, and if the last pointer stays $null$, $Tail$ still points to the end :

$$\left.\begin{array}{c} \boxminus((R_{tid} \vee G_{tid}) \wedge I) \\ (Tail.ptr + 1 = L \wedge [L] = N) \gg [L] = N \\ N.ptr = null \wedge queue(L_l, _, _) \end{array}\right\} \Rightarrow Tail.ptr{+}1 = L = L_l[-1] \tag{2}$$

- A non-null pointer of the queue is a constant :

$$\left.\begin{array}{c} \boxminus((R_{tid} \vee G_{tid}) \wedge I) \\ (queue(L_l, _, _) \wedge (L-1) \in L_l \wedge [L] = P) \gg (queue(L'_l, _, _) \wedge (L-1) \in L'_l) \\ P.ptr \neq null \end{array}\right\} \Rightarrow [L] = P \tag{3}$$

- If the $Head$ pointer is a constant and different from $Tail$, the queue continues to grow :

$$\left.\begin{array}{c} \boxminus((R_{tid} \vee G_{tid}) \wedge I) \\ H = Head \gg H.ptr \neq Tail.ptr \gg N = H.ptr \rightarrow next \gg H = Head \end{array}\right\} \Rightarrow \left\{\begin{array}{c} Heap.ptr \neq Tail.ptr \\ N = H.ptr \rightarrow next \end{array}\right. \tag{4}$$

We now give a *Hoare* triple in sequential logic which can be used to prove each basic operation :

- *enqueue* :
  $\{queue(L_l, L_v, L_p) * l \mapsto (v, null, _) \wedge Tail.ptr = L_l[-1]\}$
  `Tail.ptr->next = (pointer)(l, Tail.ptr->next.c + 1);`
  $\{queue(L_l + [l], L_v + [v], L_p[0..-2] + [(l, L_p[-1].c{+}1), (null, _)])\}$

- *dequeue* :
  $\{queue(l : L_l, v : L_v, p : L_p) \wedge Head.ptr \neq Tail.ptr\}$
  `Head = (pointer)(Head.ptr->next.ptr, Head.c + 1);`
  $\{queue(L_l, L_v, L_p) * L_l[0] \mapsto (L_v[0], L_p[0].ptr, L_p[0].c)\}$

- *changetail* :
  $\{queue(L_l, L_v, L_p) \wedge Tail.ptr \rightarrow next.ptr \neq null\}$
  `Tail = (pointer)(Tail.ptr->next.ptr, Tail.c + 1);`
  $\{queue(L_l, L_v, L_p)\}$

6.4.2 Proof

We will use the notation $\boxed{P}$ for $(P * true) \wedge I$, *i.e.*, a property about the invariant data structure. Note that we sometimes use temporal operators in beginning of atomic blocks, but we could always reformulate proofs such that they are not necessary (with an imply).

```
void enqueue(int value) {
```
$\{\boxed{true}\}$
```
01:  node * node;
02:  node->value = value;
03:  node->next.ptr = NULL;
```

$\{\boxed{true} * node \mapsto (value, null, _)\}$

```
04:  while(true) {
05:    tail = Tail;
```

$\{\boxed{\diamondminus tail = Tail} * node \mapsto (value, null, _)\}$

```
06:    next = tail.ptr->next;
```

$\{\boxed{(\diamondminus tail = Tail) \wedge (\diamondminus next = tail.ptr \rightarrow next)} * node \mapsto (value, null, _)\}$

```
07:    if(tail == Tail) {
```

by (1), with $p = Tail$ and $P = tail$, since $Tail = tail$:

$\{\boxed{P_{tail,next}} * node \mapsto (value, null, _)\}$

with $P_{tail,next} = \diamondminus(tail = Tail \wedge next = Tail.ptr \rightarrow next)$.

```
08:      if(next.ptr == NULL) {
```

$\{\boxed{next.ptr = null \wedge P_{tail,next}} * node \mapsto (value, null, _)\}$

```
09:        /* if(CAS(&tail.ptr->next, next, (pointer)(node, next.c+1))) */
09:        b = false;
09:        <if(tail.ptr->next == next) {
```

$\{\boxed{tail.ptr \rightarrow next = next \wedge next.ptr = null \wedge P_{tail,next}} * node \mapsto (value, null, _)\}$

by (2) and $P_{tail,next}$, with $L = tail.ptr + 1$ and $N = next$:

$\{\boxed{tail.ptr = Tail.ptr = L[-1] \wedge tail.ptr \rightarrow next = next} * node \mapsto (value, null, _)\}$

```
09:          tail.ptr->next = (pointer)(node, next.c+1);
```

At this point we just achieved an *enqueue* operation and :

$\{\boxed{node \neq null \wedge tail.ptr \rightarrow next.ptr = node \wedge Tail.ptr = tail.ptr}\}$

```
09:          b = true;
09:        }>
09:        if(b) {
```

$\{\boxed{node \neq null \wedge \diamondminus(tail.ptr \rightarrow next.ptr = node \wedge Tail.ptr = tail.ptr)}\}$

```
10:          /* CAS(&Tail, tail, (pointer)(node, tail.c+1)); */
10:          <if(Tail == tail)
```

$\{\boxed{node \neq null \wedge Tail = tail \wedge \diamondminus(tail.ptr \rightarrow next.ptr = node \wedge Tail.ptr = tail.ptr)}\}$

then by (3), with $L = tail.ptr{+}1$ and $P = (node, _)$, since $Tail = tail$:

$\{\boxed{node \neq null \wedge Tail = tail \wedge Tail.ptr \rightarrow next.ptr = node}\}$

```
10:            Tail = (pointer)(node, tail.c+1);
```

$Tail$ pointer has been advanced, which is the *changetail* operation.

```
10:          >
11:          return;
12:        }
13:      } else
14:        /* CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1)); */
15:        <if(Tail == tail)
```

$\{\boxed{Tail = tail \wedge next.ptr \neq null \wedge P_{tail,next}} * node \mapsto (value, null, _)\}$

by (3) with $L = tail.ptr + 1$ and $P = next$, since $Tail = tail$:

$\{\boxed{Tail = tail \wedge next.ptr \neq null \wedge Tail.ptr \rightarrow next.ptr = next} * node \mapsto (value, null, _)\}$

```
15:          Tail = (pointer)(next.ptr, tail.c+1);
```

$Tail$ pointer has been advanced, which is the *changetail* operation.

```
15:        >
16:  }
```

```
}

bool dequeue(int * pvalue) {
```
$\{\boxed{true}\}$
```
01:  while(true) {
02:    head = Head;
03:    tail = Tail;
04:    next = head.ptr->next;
```
$\{\boxed{(head = Head \blacktriangleright tail = Tail \blacktriangleright next = head.ptr \rightarrow next) \wedge (\diamondminus head = Head) \wedge (\diamondminus next = head.ptr \rightarrow ne}$
```
05:    if(head == Head) {
```
by (1), with $p = Head$ and $P = head$, since $Head = head$:

$\{\boxed{(head = Head \blacktriangleright tail = Tail \blacktriangleright next = head.ptr \rightarrow next) \wedge P_{head,next}}\}$

with $P_{head,next} = \diamondminus(head = Head \wedge next = head.ptr \rightarrow next)$.
```
06:      if(head.ptr == tail.ptr) {
07:        if(next.ptr == NULL)
08:          return FALSE;
```
$\{\boxed{head.ptr = tail.ptr \wedge next.ptr \neq null \wedge P_{head,next}}\}$
```
09:        /* CAS(&Tail, tail, (pointer)(next.ptr, tail.c+1)) */
09:        <if(Tail == tail) {
```
$\{\boxed{Tail = tail \wedge head.ptr = tail.ptr \wedge next.ptr \neq null \wedge P_{head,next}}\}$

by (1), with $p = Tail$ and $P = tail$, since $head.ptr = tail.ptr$:

$\{\boxed{Tail = tail \wedge next.ptr \neq null \wedge P_{tail,next}}\}$

with $P_{tail,next} = \diamondminus(tail = Tail \wedge next = Tail.ptr \rightarrow next)$.

Then, by (3) with $L = tail.ptr + 1$ and $P = next$:

$\{\boxed{Tail = tail \wedge next.ptr \neq null \wedge next = Tail.ptr \rightarrow next}\}$
```
09:          Tail = (pointer)(next.ptr, tail.c+1);
```
Tail pointer has been advanced, which is the *changetail* operation.
```
09:        >
10:      } else {
11:        *pvalue = next.ptr->value;
```
$\{\boxed{(head = Head \blacktriangleright tail = Tail \blacktriangleright next = head.ptr \rightarrow next) \wedge head.ptr \neq tail.ptr}\}$
```
12:        /* if(CAS(&Head, head, (pointer)(next.ptr, head.c+1))) { */
12:        b = false;
12:        <if(Head == head) {
```
since $head.ptr \neq tail.ptr$:

$\{\boxed{(head = Head \gg head.ptr \neq Tail.ptr \gg next = head.ptr \rightarrow next \gg head = Head)}\}$

by (4), with $H = head$ and $N = next$:

$\{\boxed{Head.ptr \neq Tail.ptr \wedge next = Heap.ptr \rightarrow next \wedge head = Head}\}$
```
12:          Head = (pointer)(next.ptr, head.c+1);
```
Hence we did a *dequeue* operation and :

$\{\boxed{true} * head.ptr \mapsto (_,_,_)\}$
```
12:          b = true;
12:        }>
12:        if(b) {
```
$\{\boxed{true} * head.ptr \mapsto (_,_,_)\}$
```
13:          free(head.ptr);
```

$\{\boxed{true}\}$

```
13:          return TRUE;
14:        };
14:      }
15:    }
16:  }
}
```

6.5 Real implementations

Trying to prove this algorithm made me understand that it is actually really tricky, so I wanted to check if real implementations are sounds :

- http://www.codeproject.com/KB/cpp/lockfreeq.aspx is wrong (l.6 non-atomic in *enqueue*, dereferencing errors in C++ version, ...)
- Java version in ClassPath seems sound (and simpler, thanks to a GC)
- original article implementation seems sound, using a specific memory heap manager so dereferencing never fails
- Iris library seems sound, but they use a pointer counter modulo only 4, so the probability of an overflow is hight

6.6 Future work

The next goal will be to formalize this proof in *Coq* once we will get a good framework for this logic.

There is also a improved algorithm by Ladan-Mozes and Shavit, more complex but with an interesting approach, which should be verified too.

Then, we could remove the assumptions on the dereferencings, using an explicit free memory pool and `malloc` and `free` functions. The algorithm used must enforce the memory space to grow, like in the ROP paper.

7 *Coq* implementation

The goal is to give a Coq formalization of Fu's logic to verify the soundness of the logic and use it to prove several algorithms.

I worked and continue to work with Yanni Kouskoulas. The code is available on a public Mercurial repository : http://bitbucket.org/guillaumeclaret/fu_logic/.

7.1 Code base and first improvements

This project will follow a previous formalization attempt I stopped three weeks before, so there is already a code base. But I get stuck on many problems we will try to solve here.

7.1.1 Other separation logic implementations

- `http://staff.aist.go.jp/reynald.affeldt/seplog/` : project to verify pieces of MIPS assembly using separation logic ; very complete, with concrete examples
- Tactics for Separation Logic : a paper to present a general way to implement good separation logic tactics

7.1.2 Syntax trees of assertions

There is the choice of describing the assertions first as a syntax, with an abstract tree and an evaluation function to give the semantics, or directly as Coq assertion on states or traces.

The first possibility allows more automation, since the syntax allows introspection to do substitutions, which can be critical in rules for assignments. For example, if we want to compute the *Weakest Precondition* of P for a store variable assignment :

$$\{WP(P) = P[v := E]\}\ \texttt{v = E;}\ \{P\}$$

we need to do a substitution on the variable v in P.

But if we represent predicates as Coq functions of type `state -> Prop`, it is impossible to compute such an explicit form. We can only do a function composition for the general case :

$$WP(P) = P \circ \{(s,h) \mapsto (s[v := [\![E]\!]_{(s,h)}], h)\}$$

and next we need to apply some tactics, based on rewriting rules, to get an explicit result. What is both more complex and general than the previous form.

However, it is often much easier to manipulate, because syntax and semantics are the same, easier to extend with new predicates. It also allows the usage of Coq operators, like `forall` or `exists`, without having to define the notion of assertion variable, free variable, ... which can be a mess to formalize.

7.1.3 Logical variables

We often want to use logical variables in assertions to record the value of a program expression at a special point during the execution. For example, with X a logical variable to record the value of E :

$$\frac{}{\{X = E\}\ v = E;\ \{v = X\}}$$

The question is how to define precisely X.

We could first introduce a code annotation mechanism, with a `let` pseudo-instruction. For that, we need to extend the program state with a store for logical variables, ie the state become a triple $(store, heap, logical_variables)$. The semantics of a `let` is then to assign a new logical variable to the value of an expression at a program

point. It allows us to deduce the rule :

$$\frac{\{P \wedge X = E\}\ C\ \{Q\}}{\{P\}\ \textbf{let}\ X = E\ \textbf{in}\ C\ \{Q\}}$$

so we can always enrich the hypothesis with the value of an expression.

But this method need to extend artificially the program state. Actually, we do not need it, if we use an external universal quantification instead ; for example, for the assign rule :

$$\frac{}{\forall X \in int,\ \{X = E\}\ v = E;\ \{v = X\}}$$

This quantification is not part of the assertion language. To remove it, we have this simple rule :

$$\frac{\forall X \in int,\ \{P \wedge X = E\}\ C\ \{Q\}}{\{P\}\ C\ \{Q\}}$$

where P and Q do not depend on X.

7.1.4 Equivalence of assertions

We will have to manipulate assertions all the time doing proofs, doing simplifications or term reorganization on them. So we need an easy way to claim and to show that two assertions are equivalent :

$$P \sim Q \iff \forall s \in State,\ P(s) \Leftrightarrow Q(s)$$

This is the natural definition, and we can next show basic results such as :

$$emp * P \sim P$$

$$P * Q \sim Q * P$$

But, since it is just an equivalence class, we would need to show that all our operations using assertions are congruences according to $\sim$, to be able to do replacements. Fortunately, this can be avoided adding two logical axiom to Coq :

- proposition's extensionality : $P \Leftrightarrow Q \implies P = Q$
- functional extensionality : $\forall x, f(x) = g(x) \implies f = g$

Using this, we can then show :

$$P \sim Q \implies P = Q$$

and we get that all the operations are congruences for free.

7.1.5 Non-empty traces

Traces have to get at least one initial state, so they are represented as non-empty lists. A trace is then a couple, a list and a proof that the list is not empty.

7.2 Future work

Since the basis of the formalization are well established, it should be possible to prove easily the inference rules which are not yet checked.

To get a usable framework to formalize programs' proofs, an important work will also have to be done on proof automation, with a set of tactics to easily handle operations such as term reordering, or term simplification.

8 Conclusion

In this work I successfully proved an example of optimistic algorithm with a new way, which significantly simplify or increase the formalization of previous well-known proofs.

I also started a *Coq* project, which helped to make clear some details of the historical logic used. Furthermore, it is intended to be used in industry, since people working on surgery robots are interested and contributing to this project in order to check their algorithms.

9 Acknowledgments

I thanks my advisor, the professor *Zhong Shao*, for his knowledge and his advices, the Yale University which hosted me.

I also thanks people I worked with, *Ming Fu* and *Yannis Kouskoulas*, for the experience we shared. And all the people who welcomed me in America for this unforgivable internship.

References

[1] Chlipala. *Syntactic Proofs of Compositional Compiler Correctness.* PhD thesis.

[2] Ming Fu. *Reasoning about Optimistic Concurrency Using a Program Logic for History.* PhD thesis, 2010.

[3] Xavier Leroy. *Formal verification of a realistic compiler.* PhD thesis, 2009.

[4] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms.

[5] Zhaozhong Ni, Xinyu Feng, and Zhong Shao. *An Open Framework for Foundational Proof-Carrying Code.* PhD thesis.

[6] Peter W. O'Hearn. *Local Action and Abstract Separation Logic.* PhD thesis, 2007.